NumScout: Unveiling Numerical Defects in Smart Contracts Using LLM-Pruning Symbolic Execution

Jiachi Chen, *Member, IEEE*, Zhenzhe Shao, Shuo Yang, Yiming Shen, Yanlin Wang, Ting Chen, *Member, IEEE*, Zhenyu Shan, and Zibin Zheng, *Fellow, IEEE*

***Abstract*—In recent years, the Ethereum platform has witnessed a proliferation of smart contracts, accompanied by exponential growth in total value locked (TVL). High-TVL smart contracts often require complex numerical computations, particularly in mathematical financial models used by many decentralized applications (DApps). Improper calculations can introduce numerical defects, posing potential security risks. Existing research primarily focuses on traditional numerical defects like integer overflow, and there is currently a lack of systematic research and effective detection methods targeting new types of numerical defects. In this paper, we identify five new types of numerical defects through the analysis of 1,199 audit reports by utilizing the open card method. Each defect is defined and illustrated with a code example to highlight its features and potential consequences. We also propose NumScout, a symbolic execution-based tool designed to detect these five defects. Specifically, the tool combines information from source code and bytecode, analyzing key operations such as comparisons and transfers, to effectively locate defects and report them based on predefined detection patterns. Furthermore, NumScout uses a large language model (LLM) to prune functions which are unrelated to numerical operations. This step allows symbolic execution to quickly enter the target function and improve runtime speed by 28.4%. We run NumScout on 6,617 real-world contracts and evaluated its performance based on manually labeled results. We find that 1,774 contracts contained at least one of the five defects, and the tool achieved an overall precision of 89.7%.**

***Index Terms*—Smart contracts, numerical defects, LLM, symbolic execution.**

I. Introduction

SINCE the launch of Ethereum [1] in 2015, smart contracts have emerged as a key technology in the blockchain space. Smart contracts are computer programs that automatically enforce predefined agreements on the blockchain, executing transactions without requiring intermediaries. With the rapid development of the Ethereum ecosystem, the number of smart contracts on Ethereum and other blockchain platforms has grown significantly, giving rise to numerous token contracts and decentralized applications (DApps) [2]. Meanwhile, the digital assets involved in these contracts and applications have grown exponentially.

During the design and development of smart contracts, developers frequently handle various numerical computations. In particular, many DApps rely on mathematical financial models that require highly complex computations [3]. However, owing to the characteristics of the Solidity programming language [4] and the inherent limitations of blockchain platforms, smart contracts are susceptible to various numerical defects. In this paper, we define numerical defects as all numerical-related *errors, vulnerabilities, or flaws* that can lead to unexpected outcomes or deviate from the original code's intent [5]. Notably, numerical defects involve not only security issues but also design flaws, which can increase the long-term risk of the smart contracts.

Numerous real-world hacking incidents caused by numerical defects have already resulted in severe financial losses for both project teams and users. Although common numerical security defects, such as integer overflow and type conversion errors [6], have been identified and mitigated through solutions like the SafeMath library [7] and the introduction of new security mechanisms in Solidity v0.8 [8], new types of numerical defects continue to emerge in practice. For example, over $2.12 million in assets were stolen from *Balancer* [9] due to a precision-related issue [10]. These numerical defects pose significant threats to the security and reliability of contracts. However, a systematic study that classifies new types of numerical defects

Received 6 January 2025; revised 12 March 2025; accepted 18 March 2025. Date of publication 28 March 2025; date of current version 16 May 2025. This work was supported in part by Zhejiang Provincial Key Project of Undergraduate Education and Teaching Reform under Grant JGZD2024060, in part by Zhejiang Provincial Higher Education Research Project & Special Research Project on Artificial Intelligence Empowering Education and Teaching Applications under Grant KT2024007, in part by Sichuan Provincial Natural Science Foundation for Distinguished Young Scholars under Grant 2023NSFSC1963, and in part by the National Natural Science Foundation of China under Grant 62332004. Recommended for acceptance by G. Fraser. *(Corresponding author: Zhenyu Shan.)*

Jiachi Chen is with the Intelligent Transportation and Information Security Laboratory, Hangzhou Normal University, Hangzhou 311121, China, and also with the School of Software Engineering, Sun Yat-sen University, Zhuhai 519082, China (e-mail: chenjch86@mail.sysu.edu.cn).

Zhenzhe Shao, Shuo Yang, Yiming Shen, Yanlin Wang, and Zibin Zheng are with the School of Software Engineering, Sun Yat-sen University, Zhuhai 519082, China (e-mail: shaozhzh3@mail2.sysu.edu.cn; yangsh233@mail2.sysu.edu.cn; shenym7@mail2.sysu.edu.cn; wangylin36@mail.sysu.edu.cn; zhzibin@mail.sysu.edu.cn).

Ting Chen is with the School of Computer Science and Engineering, School of Cyber Security, University of Electronic Science and Technology of China, Chengdu 611731, China, and also with Kashi Institute of Electronics and Information Industry, Kashi 844000, China (e-mail: brokendragon@uestc.edu.cn).

Zhenyu Shan is with the Intelligent Transportation and Information Security Laboratory, Hangzhou Normal University, Hangzhou 311121, China (e-mail: 20100119@hznu.edu.cn).

Digital Object Identifier 10.1109/TSE.2025.3555622

0098-5589 © 2025 IEEE. All rights reserved, including rights for text and data mining, and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Hangzhou Normal University. Downloaded on July 22,2025 at 02:27:38 UTC from IEEE Xplore. Restrictions apply.

and provides corresponding detection methods and tools is still lacking.

To fill the gap, we first conducted an empirical study to define new types of numerical defects by analyzing 1,199 audit reports using an open card sorting method [11]. Based on this analysis, we identified five categories of new numerical defects, i.e., *Div In Path*, *Operator Order Issue*, *Minor Amount Retention*, *Exchange Problem*, and *Precision Loss Trend*. We present examples for each defect type and propose corresponding mitigation strategies to enhance the quality and robustness of smart contracts.

Then, we developed a tool named NumScout, designed to detect the five new types of numerical defects in real-world contracts. NumScout leverages the reasoning capabilities of Large Language Models (LLMs) [12] and combines source code level information with bytecode analysis to enhance detection efficiency in complex contracts. Specifically, NumScout first uses LLM-based pruning to exclude functions unrelated to numerical operations or token transfers. This step is designed to mitigate the path explosion problem in symbolic execution and accelerate the analysis process of the tool. Due to the complex semantics and call relationships of contracts, static pruning methods based on simple rule matching fail to meet the requirements. LLMs can perform reasoning and analysis at the high-level semantic layer and across multi-level calls. By leveraging a multi-role collaboration strategy, they reduce response randomness and error, thereby accomplishing the pruning task effectively. Then, based on predefined patterns and a symbolic execution framework, the tool performs symbolic execution at the bytecode level, incorporating features from the source code for further analysis. It focuses on key operations such as comparisons and transfers, and identifies defects through various methods, including constructing and analyzing expression operator order trees, extracting comparison statements from bytecode, and analyzing token flows.

To demonstrate the prevalence of the five defined numerical defects and evaluate the efficacy of NumScout, we filter 6,617 real-world smart contracts which are frequently used by users on Ethereum [13], ensuring that the contracts in our experimental dataset have actual value rather than toy contracts. We apply NumScout to these 6,617 smart contracts and find that 1,774 contracts contain at least one of the five defined defects. Then, we randomly sample contracts with a 95% confidence level and a 10% confidence interval for manual labeling. The results show that the tool achieves an overall precision of 89.7%. In addition, we conduct ablation experiments to verify the effectiveness of GPT-based pruning. The experiments demonstrate that pruning enables symbolic execution to quickly enter the target functions, improving runtime speed by 28.4% and detecting more defects.

The main contributions of our work are as follows:

- We summarize and define five new types of numerical defects based on analyzing 1,199 audit reports. For each defect, we provide its definition with a code example for better illustration. Furthermore, we outline possible solutions to enhance development security.
- We develop NumScout, the first tool designed for the defined numerical defects. NumScout employs LLM pruning functions and recovers source-level features from bytecode during symbolic execution to identify designed defect patterns more efficiently.
- We evaluate NumScout's performance on 6,617 real-world smart contracts and discover that 1,774 contracts contain at least one defined defect. Moreover, in a manually labeled dataset created through random sampling, our approach achieves an overall precision of 89.7%.
- We make the source code of NumScout, all experimental data, and analysis results publicly available, along with detailed Markdown files at https://github.com/NumScout/NumScout.

II. BACKGROUND

A. Numerical Operations in Solidity and Integer Overflow

Solidity is the most popular programming language for smart contracts on Ethereum. The computations in Solidity smart contracts are performed using arithmetic opcodes, e.g., *ADD* and *MUL* [14]. Due to the inherent characteristics of the language and the limitations of the blockchain platform, for example, maintaining the consistency of the public ledger and reducing computational resource consumption, Solidity only supports integers and does not support floating-point numbers, which can introduce certain numerical issues. In traditional numerical detection, integer overflow is one of the most common defects in smart contracts [15], [16], [17], [18]. An integer overflow defect occurs when the result of an arithmetic operation exceeds the range of its data type, producing an outcome that deviates from expectations. Since smart contracts typically use integers to represent asset amounts and other numerical values, calculations involving these numbers may experience overflow or underflow under malicious input from attackers, resulting in asset loss. Several notable attacks have occurred due to this defect, including *BeautyChain* token (BEC) [19] attack, *SmartMesh* token (SMT) [20] attack, and *UselessEthereumToken* token (UET) [21] attack.

The developer community has built security libraries to prevent overflows, such as the widely adopted SafeMath library [7], developed by the well-known blockchain security team OpenZeppelin [22], which ensures the correctness of calculation through boundary checks. Starting from version v0.8.0, the Solidity compiler introduces arithmetic checking mechanisms [8], which embed overflow detection into the compiled bytecode. If an overflow occurs during a transaction, the EVM [1] will throw an error and revert. However, although traditional integer overflows have been largely mitigated, increasingly complex contract scenarios are giving rise to new types of numerical defects that are easily overlooked.

B. Smart Contract Audit Report

Smart contract auditing is an important process in the blockchain ecosystem, focusing on identifying vulnerabilities and defects in smart contract code. Auditors from professional auditing teams assess the code to identify potential defects, ensuring that the contract operates as intended and adheres to

Authorized licensed use limited to: Hangzhou Normal University. Downloaded on July 22,2025 at 02:27:38 UTC from IEEE Xplore. Restrictions apply.

best practices. Audit reports provide a comprehensive analysis of smart contracts, detailing all identified defects and their impacts, assigning severity levels, and offering recommended remediation strategies. These reports serve as essential documentation for developers, investors, and users, enhancing the transparency and trustworthiness of the project's contracts. Given the irreversibility of blockchain transactions, thorough auditing is vital to prevent financial losses and maintain the integrity of DApps.

C. Large Language Models

Large Language Models (LLMs) [23], [24] are deep learning-based natural language processing models that possess powerful language understanding and generation capabilities. The GPT (Generative Pre-trained Transformer) series, developed by OpenAI [25], is a prominent representative of LLMs. GPT utilizes the Transformer [26] architecture and is trained on extensive corpora, including source code descriptions of various programming languages and known defects. With this knowledge, GPT can understand and interpret source code, enabling zero-shot learning [27]. The latest version, GPT-4o [28], supports a 128k context length, making it suitable for complex and multi-step tasks. While LLMs and GPT have shown significant potential in fields such as smart contract analysis, trustworthiness and accuracy remain critical research challenges [29], [30].

Multiple studies have demonstrated that LLMs exhibit excellent code understanding capabilities. They have great ability in understanding code syntax and semantics, including Abstract Syntax Tree (AST) and Control Flow Graph (CFG) [31]. LLMs have been applied in multiple fields that require code understanding [32]. For example, they are used for analyzing inconsistencies in code comments [33]. They also serve as the foundation for developer assistance tools [34]. Furthermore, in the field of smart contract vulnerability detection, LLMs act as code understanding tools to identify logical vulnerabilities [35], [36], [37], [38], [39].

D. Symbolic Execution

Symbolic execution-based defect detection for smart contracts primarily involves symbolizing the storage variables and external inputs within the contract. Smart contracts are typically executed on the Ethereum Virtual Machine (EVM). The EVM features a stack-based architecture and is responsible for interpreting and executing the opcodes of contracts. To describe the execution flow of contracts more clearly, the Control Flow Graph (CFG) is often utilized. The CFG represents the program's basic blocks and their control flow relationships, aiding the analysis of the reachability of different execution paths. During the symbolic execution process, a set of path constraints is maintained for each explored execution paths. These constraints consist of conditions related to symbolic variables, which describe the current execution state of the contract. The satisfiability modulo theories (SMT) solver is used to evaluate these constraints and determine whether specific conditions are satisfied, such as identifying inputs that may trigger vulnerabilities or verifying the solvability of constraints after adding new conditions. The typical workflow of symbolic execution tools is as follows: Execute the program's opcodes sequentially, symbolize variables and external inputs as they are encountered, update the program context state and add new conditions to path constraints during the process. While exploring all the executable paths of the program, the tool assesses the satisfiability of security-related conditions to detect potential security issues.

Traditional symbolic execution methods often encounter the path explosion problem, which can prevent the completion of detection within a reasonable timeframe. To address this issue, pruning methods are necessary to mitigate path explosion and accelerate the analysis process. As the complexity of smart contracts has increased in recent years, particularly in terms of semantics and call relationships, traditional pattern-based pruning methods tend to be less effective. In contrast, LLMs can recognize high-level semantics and multi-level calls, making them well-suited for completing the pruning tasks of complex contracts.

III. New Numerical Defects

In this section, we explain how the five new types of numerical defects are identified and provide definitions and examples for each defect.

A. Data Source

To identify and define new types of numerical defects, we analyze 1,199 audit reports collected by DAppScan [40]. DAppScan is a public dataset containing audit reports collected from the official websites, social media, and Web3 sites of 29 well-known blockchain security teams, such as Openzeppelin [22] and Consensys [41]. These audit reports serve as a rich resource, revealing numerous numerical defects found in real-world projects. We adopt a keyword matching approach to filter reports content related to numerical defects, while employing Snowball Sampling [42] strategy to ensure the completeness of the keyword list. Initially, we filter the audit reports by matching the keywords "precision" and "rounding". During the review of the report content, we record new keywords related to numerical defects and add them to the keyword list for filtering new reports. Ultimately, we filter a total of 194 audit reports using 25 keywords for further analysis. For the complete keyword list, please refer to our online repository.

B. Audit Report Analysis

1) Manual Filtering: In the previous subsection, we describe the collection of 194 publicly available audit reports from renowned blockchain security teams. However, some of these reports are not directly related to numerical defects. For example, certain reports mention "precision loss" but only discuss its risk or offer general advice to users, instead of detailing specific defects in the code. Therefore, we manually remove reports that lack specific defect descriptions. After filtering, we find 109 reports directly related to numerical defects from the initial set of 194 security reports.

Authorized licensed use limited to: Hangzhou Normal University. Downloaded on July 22,2025 at 02:27:38 UTC from IEEE Xplore. Restrictions apply.

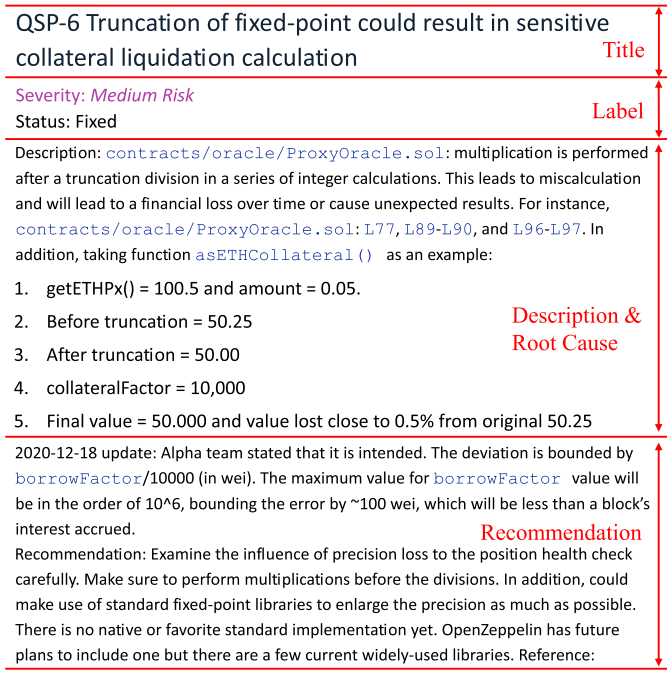

Fig. 1. Example of a card of audit reports.

2) Open Card Sorting: To ensure accuracy, we use the open card sorting [11] approach to analyze and categorize the filtered audit reports related to numerical defects. In this process, we consider two aspects to ensure the representativeness and significance of the defects, i.e., the reproducibility of the code issue and severity as assessed by the security teams. Some issues may be tightly coupled with specific applications and not reproducible; we do not classify these as representative defects. Additionally, we focus on the labels assigned by security teams in the reports to assess the severity of the identified defects.

For each numerical defect mentioned in the audit reports, we create a card comprising four sections to organize the content. Following the detailed steps outlined in [43], we begin by randomly selecting 40% of the cards for the first round of classification. First, we read the titles and descriptions of the reports to understand the relevant defects. Next, we inspect the problematic code to identify the root cause and cross-reference it with the audit reports. Finally, we review the recommended solutions suggested by the security teams to understand how to address the defects and record the severity level assigned by the team.

In the second round of classification, two authors independently categorize the remaining 60% of the cards following the same steps described in the first round. We then compare their results and discuss discrepancies. Next, we remove uncommon defects and ultimately classify the remaining defects into five types. Among the classified reports, 12 are labeled as high, 33 as medium, and 75 as low severity.

Fig. 1 shows a card example of an audit report describing a numerical defect. The card contains a title, assigned label, description, root cause, and recommended solution. From the report, we learn that the contract contains a defect where division is incorrectly performed before multiplication. We then locate the referenced code (i.e., contracts/oracle/ProxyOracle.sol: L77, L89-L90, and L96-97) to further confirm the root cause and verify the presence of this defect. The report also provides an example of miscalculations caused by this defect, demonstrating its exploitability and the potential consequences. Due to the reproducibility of this defect and its frequent occurrence in audit reports, we classify it as a distinct defect type named "*Operator Order Issue*".

TABLE I
DEFINITIONS OF THE FIVE DEFECTS

Contract Defect	Definition
Div In Path	The use of division in comparison conditions affects the execution path.
Operator Order Issue	Dividing before multiplying amplifies precision loss.
Minor Amount Retention	When multiple parties share profits, indivisible amounts remain trapped in the contract and cannot be withdrawn.
Exchange Problem	Errors in token amount calculations during token exchanges create rounding issues or profit opportunities.
Precision Loss Trend	Incorrect rounding methods lead to unreasonable allocation of precision loss.

C. Defects Definition

Based on the analysis of the audit reports mentioned in the previous section, we have summarized five new types of numerical defects. Table I provides a brief definition of each defect, followed by a detailed definition and code example for each defect pattern.

(1) Div In Path. The Solidity programming language does not support floating-point numbers, so all division operations result in integer division [44]. When the result is not a whole number, only the integer part is retained, leading to precision loss. If division is used within a conditional statement, this inherent precision loss can potentially alter the program's execution path and cause unexpected results. Consequently, users may be misled by this defect and pass incorrect values. The blockchain security team ChainSecurity [45] issues a warning about this defect in their audit report for the *Angle Protocol Borrowing Module* project.

Example: As shown in Fig. 2, users can purchase tokens by sending ether through the $getTokens$ function. The internal conditional check restricts the minimum amount of ether, with the $minAmount$ set to 3. However, users who are not familiar with Solidity may assume that sending more than 3 $ether$ will satisfy the condition, and the program will enter the expected execution branch for token purchase. In reality, the requirement is that $msg.value$ must exceed 4 $ether$. For amount between 3 $ether$ and 4 $ether$, e.g., 3.5 $ether/1$ $ether = 3$ instead of 3.5, this condition is still not met, preventing users from buying tokens. If the contract does not handle such situations, users

Authorized licensed use limited to: Hangzhou Normal University. Downloaded on July 22,2025 at 02:27:38 UTC from IEEE Xplore. Restrictions apply.

```
function getTokens(address _to, uint256 _amount)
    public payable returns (bool) {
    if(msg.value / 1 ether > minAmount) {
        /* buy tokens */}}
```

Fig. 2. An example of Div In Path defect.

```
function updatePool(uint256 _pid) public {
    // deduct 10% for the developers
    uint256 devReward = almReward.div(100).mul(10);
    _safeAlmTransfer(devaddr, devReward);}
```

Fig. 3. An example of operator order issue defect.

may lose their funds without receiving any tokens. Malicious contracts can exploit this defect to scam inexperienced users.

(2) Operator Order Issue. The most common defect regarding calculation order is performing division before multiplication. This defect leads to incorrect calculation results because multiplication can amplify the precision loss introduced by division. Therefore, in programming practices, when both multiplication and division appear in an expression, it is generally recommended to perform multiplication first and then division to minimize precision loss. However, in today's increasingly complex contracts, developers often overlook this principle, and cases where division is done before multiplication frequently occur. *Operator Order Issue* is also the most common defect in audit reports. The security team QuillAudits [46] includes a warning about this defect in their audit report for the *Alium Finance Smart Contract* project.

Example: Fig. 3 shows the $updatePool$ function, which handles the logic for retrieving staking rewards $almReward$, with 10% of the rewards allocated to the developer, i.e., $devReward$. This calculation results in a precision loss of one decimal. If $almReward = 199$, then $devReward = (199/100) * 10 = 10$. However, the calculation of 10% of $almReward$ should result in 19. If the code `devReward = almReward.mul(10).div(100)` is used instead, then the result will be correct. This defect can lead to financial losses for developers over time.

(3) Minor Amount Retention. This defect typically arises in scenarios where multiple participants share rewards or withdraw funds. On blockchain platforms, numerous game contracts involve players investing funds to participate, with winners dividing the rewards. During the distribution of rewards, if the total amount is not divisible by the number of users, a small portion of the funds will remain stuck in the contract, unable to be withdrawn. If the withdrawn tokens are tied to a liquidity pool, leftover tokens could affect the ratio, leading to economic losses. The security team Dedaub [47] issues a warning about this defect in their audit report for the *GoodGhosting* project.

Example: The code shown in Fig. 4 comes from an investment game contract that incentivizes players to participate in the game and maintain the investment plans. Player can withdraw their funds and claim interest rewards generated within the game through the $withdraw$ function. These interest rewards are evenly distributed among all winning players. There is a

```
function withdraw() external virtual {
    // calc interest reward shared by all winners
    payout = payout.add(totalGameInterest.div(
        winners.length));
    require(IERC20(daiToken).transfer(msg.sender,
        payout),"Fail to transfer");}
```

Fig. 4. An example of minor amount retention defect.

defect where the $totalGameInterest$ may not be divisible by $winners.length$, resulting in a minor amount of funds remaining in the contract and being unable to be withdrawn. The retained amount can affect the contract's state, making actions that reference that state unsafe. Specifically, if there are other contracts associated with the $daiToken$ balance of this investment pool, such as trading pairs composed of $daiToken$ and other tokens, the retained amount can impact the ratio between the two, leading to financial security issues.

(4) Exchange Problem. Token exchanges are fundamental in various scenarios, such as purchasing tokens, providing liquidity, and trading tokens. If the numerical operations involved in the exchange process are not handled properly, issues may arise, including exchange rounding and zero-cost profit opportunities. The former results in users losing their input tokens while receiving zero output tokens. The latter allows users to obtain output tokens without providing any input tokens. The security team Trail of Bits [48] reports this defect in their audit of the *Balancer Finance* project.

Example: As shown in Fig. 5, the function $joinPool$ allows users to inject assets into the liquidity pool and receive corresponding pool shares (which is also an ERC20 [49] token, referred to as pool token). Therefore, there is a token exchange process involved here. The user inputs $poolAmountOut$ to indicate the amount of pool tokens they want to receive. Internally, the function calculates the exchange ratio based on the desired amount of pool tokens and the total amount of pool tokens. It then calculates the number of liquidity tokens that the user needs to contribute based on the current total of liquidity tokens held by the pool. During this process, users may receive pool tokens without having to contribute any liquidity tokens. The calculation method for $bmul$ is as follows $c = \frac{(a*b)+\frac{BONE}{2}}{BONE}$. Thus, the final expression for $tokenAmountIn$ is:

$$tokenAmountIn = \frac{bal * \frac{poolAmountOut}{poolTotal} + \frac{BONE}{2}}{BONE}$$

BONE is set to 10^{18}. Suppose the condition $bal * \frac{poolAmountOut}{poolTotal} < 5 * 10^{17}$ is satisfied, a quantity of $poolAmountOut$ pool tokens will be generated, while the user contributes no liquidity tokens, resulting in $tokenAmountIn = 0$. This situation occurs if the token has low liquidity or has decimals precision lower than 18, e.g., USDT [50], USDC [51], and XRP [52], which all hold high market values, have only 6 decimals. According to data from Etherscan, these three tokens are all ranked in the top 5 by market capitalization [53].

(5) Precision Loss Trend. There are three rounding methods for division: rounding down, rounding to the nearest integer, and rounding up. In Solidity, the default behavior for division

Authorized licensed use limited to: Hangzhou Normal University. Downloaded on July 22,2025 at 02:27:38 UTC from IEEE Xplore. Restrictions apply.

```
function joinPool(uint poolAmountOut, uint[]
    calldata maxAmountsIn) external _logs_ _lock_ {
    // calc swap ratio with input and poolTotal
    uint poolTotal = totalSupply();
    uint ratio = bdiv(poolAmountOut, poolTotal);
    uint bal = _records[t].balance;
    // calc amount user should contribute with ratio
    uint tokenAmountIn = bmul(ratio, bal);}
```

Fig. 5. An example of exchange problem defect.

```
function _dealWithPoolAndCollectFee(Context memory
    ctx, bool isBuy) internal returns (uint) {
    // calc transaction fee
    uint fee = amountToTaker * feeBPS / 10000;
    // calc amount user gets after deducting fee
    amountToTaker -= fee;
    _transferToken(token, ctx.order.sender,
        amountToTaker, ctx.isLastSwap);
    return amountToTaker;}
```

Fig. 6. An example of precision loss trend defect.

is rounding down, which returns the largest integer less than or equal to the exact division result, denoted as $floor(x)$. Rounding to the nearest integer adjusts the division result up or down based on the decimal place, and in formulas, it is expressed by adding $(denominator/2)$ to the $numerator$, denoted as $round(x)$. Rounding up returns the smallest integer greater than or equal to the normal division result, expressed as adding $(denominator - 1)$ to the $numerator$, denoted as $ceil(x)$. Using different rounding methods can cause different tendencies in the calculation results, and incorrect tendencies can lead to unexpected consequences. The security team PeckShield's [54] audit report on *OneSwap* includes this defect.

Example: In Fig. 6, the function $_dealWithPoolAndCollectFee$ is responsible for handling transactions within the trading pool and collecting fees. At this point, the fee calculation uses the standard $floor(x)$ method, rounding the result down. The tokens amounts user receives is the total amount minus the fee, and user obtains the small fractional amount discarded during the rounding down process. This means the calculation tends to favor users, allowing users to receive more tokens. However, in AMM-based DEX scenarios [55], the calculation should favor the liquidity pool to protect the interests of liquidity providers. Therefore, the fee should be rounded up, ensuring that more tokens remain in the liquidity pool. Use `fee = (amountToTaker * feeBPS + 9999) / 10000` to replace the original code.

Another scenario affected by this defect is unfair distribution of benefits as shown in the Fig. 7 below. At this point, BB receives a rounded-up value, while AA corresponds the rounded-down value. It is crucial to carefully consider the tendency of precision loss and apply the most suitable rounding method for different scenarios.

IV. METHODOLOGY

In this section, we introduce the methods for detecting the aforementioned defects. We first provide an overview of our approach, followed by detailed explanations of two main components: GPT-based pruning and symbolic execution. For the latter, we further elaborate on instruction-level details and operational features.

```
function _updatePrices() internal {
    // calc AA's earnings based on ratio
    AAGain = gain * trancheAPRSplitRatio /
        FULL_ALLOC;
    // sub AA's earnings from total to obtain BB's
    BBGain = gain - AAGain;}
```

Fig. 7. Another example of precision loss trend defect.

A. Overview

Fig. 8 presents an overview of NumScout. NumScout consists of two main components: the GPT-based function pruning and the symbolic execution detection tool. Specifically, users provide Solidity source code as input. If the code is too lengthy and exceeds the input limit of GPT, we perform subgraph segmentation. This involves analyzing the contract's abstract syntax tree (AST) and constructing a call graph starting from each entry function based on the internal call relationships. The call graph is then used to break the large contract into several smaller contracts, each with a complete function call chain. The segmented code is subsequently passed to GPT for pruning.

The pruning component of GPT involves four roles: a Classifier, two Verifiers, and a Combiner, which work collaboratively to enhance the accuracy of pruning. The Classifier generates preliminary relevance judgments based on whether the functions in the contract involve numerical operations or fund transfers, classifying them as either "related" or "unrelated" and sending the results to the two Verifiers. Each Verifier independently verifies a specific subset of the classification results. One Verifier focuses on the "related" part, ensuring that all functions classified as related are indeed associated with computations or fund transfers. The other Verifier focuses on the "unrelated" part to prevent mistakenly discarding functions that may have implicit relevance. After the verification is completed, the two Verifiers send their respective results to the Combiner. The Combiner integrates the feedback from both Verifiers and makes final adjustments to the classification results to ensure the accuracy of pruning. The pruning results are subsequently passed to the symbolic execution detection tool for further defect analysis.

The symbolic execution detection tool contains four main components [56]: the *Inputter*, *Feature Detector*, *CFG Builder*, and *Defect Identifier*. The *Inputter* accepts user-provided Solidity source code and GPT's pruning results as input. It compiles the source code using various versions of Solidity compiler to obtain the bytecode and AST, and utilizes the API provided by Geth [57] to disassemble bytecode into opcodes. The AST is analyzed to extract source mappings [58] for further analysis by other components. The *CFG Builder* performs symbolic execution and dynamically constructs the CFG while skipping the pruned function paths. It records key events (i.e., stack events, memory events, and call events) to detect defect features.

Authorized licensed use limited to: Hangzhou Normal University. Downloaded on July 22,2025 at 02:27:38 UTC from IEEE Xplore. Restrictions apply.

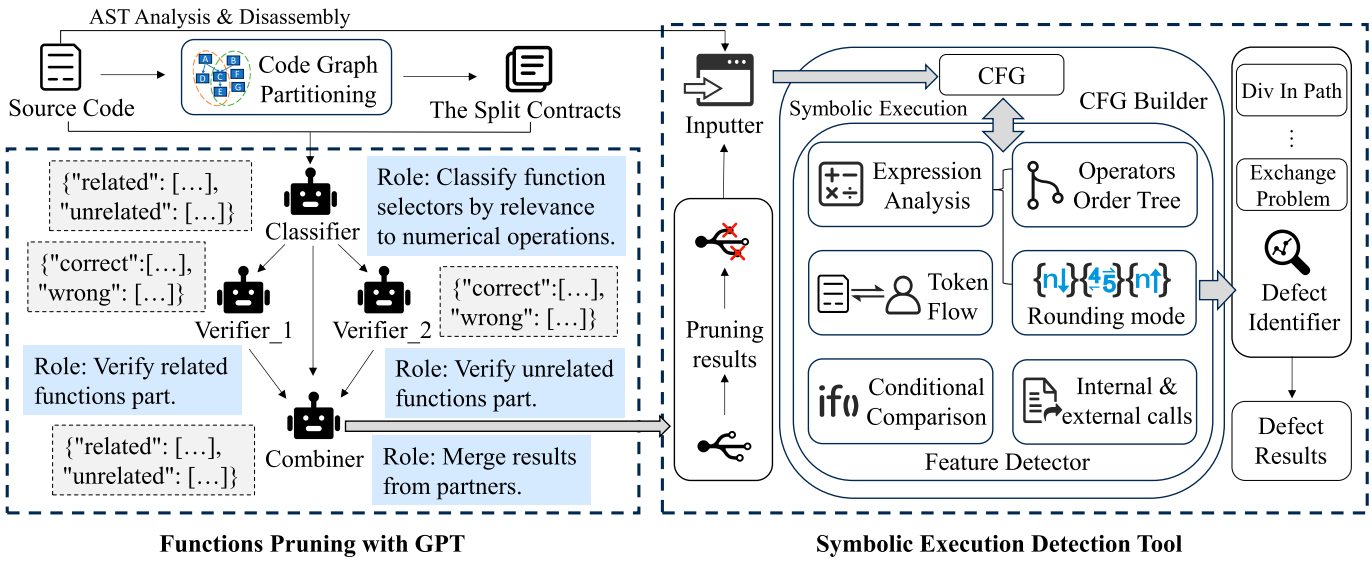

Fig. 8. An overview of the approach of NumScout.

During the CFG construction, the *Feature Detector* identifies feature operations and maintains required data structures for detection (i.e., expression information, conditional comparisons, token flows, and internal&external function calls). For each defect, the *Defect Identifier* provides the final detection results based on the predefined features with the help of satisfiability modulo theories (SMT) solver [59].

B. Functions Pruning With GPT

Motivations for Pruning. Traditional symbolic execution methods suffer from the problem of path explosion. To mitigate this problem and speed up the analysis process, we utilize LLMs for pruning by discarding entry function paths that are unrelated to numerical or transfer operations in the contract. It allows the symbolic execution framework to reach target functions more quickly.

Existing pattern-based pruning methods or manually designed heuristics struggle to handle scenarios involving complex semantics and intricate call relationships [60], [38], [37]. Take transfer operations as an example, the implementation of transfer functions in smart contracts is diverse, using different variables for accounting and may be hidden in multi-level function calls, executed conditionally based on complex logic. These complexities make it difficult for pattern-based methods to accurately capture the semantics of such operations. In contrast, as stated in Section II-C, multiple studies have demonstrated that LLMs possess powerful code understanding and reasoning capabilities and can perform outstandingly in various software engineering tasks across multiple fields [32]. LLMs can recognize complex semantics, analyze intricate calls, and determine whether a function is related to numerical operations or transfer operations from the source code level. Therefore, we use LLMs to accomplish the pruning task. For the possible inaccuracy and randomness in LLMs responses, we employ a multi-role collaboration strategy [36] to mitigate these issues. It assigns distinct roles to different LLMs, with each role focusing on a specific aspect of the analysis. They cross-verify the outputs of others to enhance overall accuracy and reliability.

The Pruning Process. Although LLMs are capable of understanding and analyzing complex semantics, their responses may exhibit hallucinations [61] and uncertainty [62]. A multi-role collaboration strategy can reduce errors and randomness in the output [36] and improve the effectiveness of pruning. The pruning process involves four roles: a Classifier, two Verifiers, and a Combiner. Considering both performance and cost, GPT-4o [28] is selected as the large language model to implement these roles. As shown in Fig. 9, to further enhance the reliability of the answers and minimize the impact of randomness in GPT's output, we adopt the "mimic-in-the-background" prompting method, which is inspired by Sun et al. [35]. With this method, GPT is prompted to simulate answering the same question five times in the background. The most frequently occurring answer is then selected to ensure higher consistency.

1) Rough Classification by Classifier: The Classifier is responsible for preliminary analysis and classification of function selectors, with the prompt template shown in Fig. 9. The prompt follows the zero-shot chain-of-thought (CoT) [27] method. CoT guides LLMs to break down complex tasks into step-by-step logical reasoning sequences when answering questions, which enhances their reasoning capabilities and improves response reliability. [%Classification Requirements%] specify that the Classifier's task is to roughly categorize function selectors based on whether they are related to token numerical operations or balance changes, dividing them into two categories: "related" and "unrelated". The function selector is a 4 byte id that Solidity uses to identify functions. [%CoT Steps%] provides specific reasoning steps to guide GPT's inference process. First, it identifies which functions perform numerical operations on token or ether amounts. Then, it examines the relationships between function calls to determine which public functions

Authorized licensed use limited to: Hangzhou Normal University. Downloaded on July 22,2025 at 02:27:38 UTC from IEEE Xplore. Restrictions apply.

System prompt		
You are a smart contract auditor. You will be asked questions related to code properties. You can mimic answering them in the background five times and provide me with the most frequently appearing answer. Furthermore, please strictly adhere to the output format specified in the question; there is no need to explain your answer.		
Classifier prompt Template	**Verifier prompt Template**	**Combiner prompt Template**
You need to answer **[%Classification Requirements%]** Think step by step: **[%CoT Steps%]** Your response: **[%Output JSON Format%]** Do not: **[%Constraints%]** All public functions: **[%Function Selectors List%]** The code I provide is: **[%Code%]**	Your responsibility is to **[%Verification Requirements%]** **# [%Abilities%]** **# [%Responsibilities%]** # **[%Constraints%]** # Classifier's result: The code I provide is: **[%Code%]**	Your responsibility is to **[%Combination Requirements%]** # **[%Constraints%]** # Classifier's result: # Verifiers_1's result: # Verifiers_2's result: The code I provide is: **[%Code%]**

Fig. 9. The prompt template used by roles.

from [%Function Selectors list%] call those numerical operation functions. Finally, it classifies them as either "related" or "unrelated". [%Constraints%] section highlights errors that GPT must avoid, e.g., "Do not add functions that are not in the list".

2) Secondary Verification by the Verifiers: Verifier_1 is responsible for verifying "related" part of the classification results from Classifier, while Verifier_2 is responsible for "unrelated" part. The results are categorized into two categories: "correct" and "wrong". The prompt templates used by both are shown in Fig. 9. Following the approach in [36], we define more detailed capabilities, responsibilities, and constraints for GPT. [%Abilities%] require GPT to act as an excellent smart contract code reviewer, familiar with function calls. [%Responsibilities%] describe the task of examining the results from the Classifier and making judgments based on those results. [%Constraints%] define the required output format and certain types of errors that are not permissible.

3) Merging Results by Combiner: Combiner synthesizes the reports from other partners to derive the final pruning results. [%Constraints%] emphasize that Combiner cannot simply merge the results but must exercise its own judgment. Additionally, it is essential for GPT to focus on the functions that highlight contradictions between the Classifier and the Verifiers when making the final determination.

C. Symbolic Execution Detection Tool

1) Operational Semantics Modeling: In this subsection, we model the syntax of several basic instructions, variables, and functions that form the foundation of the core analysis module for our detection tool.

We first present the operational semantics of two instructions related to function calls as follows. $CALL(f, o, l)$: Calls the target function f, with parameters loaded from memory starting at address o with length l. $JUMPI(c, t)$: If the jump condition c is satisfied, it jumps to location t in the program.

During symbolic execution, certain key data structures are updated according to each executed EVM instruction. Specifically, S represents the operand stack, M denotes the simulated temporary memory space, and GS represents the storage values of state variables. During path exploration, the constraints accumulated in the SMT solver are defined as the variable $cons$. Our method utilizes source-level information provided by source mappings to assist in identifying defective code locations during symbolic execution.

Expression Information Recovery. During the execution of the symbolic execution framework, expressions are simplified, resulting in the loss of expression information, i.e., operands and operators. Therefore, it is necessary to maintain a structure v to store the information before the expression simplification. To utilize the expression information, we define a function $e(a, v)$ to recursively retrieve and recover information of the given expression a from structure v, represented as $x, op, y := e(a, v)$. This allows us to analyze the original logical structure and meaning of expressions during symbolic execution.

Comparison Semantic Recognition. In Solidity, there are three types of conditional statements: if, $require$, and $assert$. The if statement enters the $else$ branch when the condition is not satisfied, whereas both $require$ and $assert$ revert the transaction and throw an exception. Compared to $require$ and $assert$, which have rollback protection and thus minimize the impact on the user in case of errors, if statements pose a greater risk as they alter the execution path. Therefore, when analyzing patterns involving division within paths, the primary focus is on if statements, making the identification of if statements crucial.

Authorized licensed use limited to: Hangzhou Normal University. Downloaded on July 22,2025 at 02:27:38 UTC from IEEE Xplore. Restrictions apply.

The key to recognizing if statements lies in identifying the comparison operator and retaining the two elements being compared. The comparison operator recognition is accomplished by matching source code with opcode sequences. Before executing the opcodes in a basic block, we first do a match process. If a corresponding opcode sequence and source code are matched, a trigger is set, and the relevant comparison operator is recorded. When executing the corresponding comparison opcode, e.g., GT or LT, the two top values on the stack are captured. At this point, the two expressions being compared are obtained, which are used for subsequent detection. We define function $cs(S)$ to retrieve the comparison elements x, y, and the comparison operator cop. The seq represents the opcode sequence of comparison operation. Once these comparison elements are identified, we can further analyze the logic behind the conditional statements and track the execution flow during symbolic execution. This is crucial for defect detection, especially in scenarios where conditional logic may lead to different execution paths.

$$x, cop, y = cs(S), \frac{S :=< x, y >, cop := match(seq)}{GT|LT, JUMPI(jc, *)}$$

External & Internal Function Calls. We focus on two types of transfer function calls: one where the token contract calls its own transfer function and another where it calls an external token contract's transfer function. For the former, the key opcode is $JUMP$, with parameters obtained from the stack. For example, when calling its own transfer function `transferFrom(from, to, amount)`, the elements on the stack are arranged from top to bottom as follows: jump destination, amount, to and from.

For the latter, the key opcode is $CALL$, which first needs to retrieve the parameters' memory locations from the stack and then access the parameters from memory. For example, when calling an external token contract's transfer function `token.transfer(to, amount)`, the seven elements on the stack, from top to bottom, are: gas consumed, token contract address, ether amount to transfer, starting memory position of parameters (o), length of parameters (l), starting memory position and length for storing the return data. Our tool reads the memory values to extract parameters based on the fourth and fifth elements. Specifically, the first four bytes at the starting memory position represent the function selector, i.e., $memory[o : o + 4]$, followed by 32 bytes for $amount$, and another 32 bytes for to.

Recognizing internal and external calls is crucial for NumScout to acquire token transfer parameters, which is essential for detecting defects related to transfer amounts. Thus, we define the function ea in Eq. 1 to retrieve the expression for the transfer amount from stack and memory.

2) Defects Detection: To detect five new types of numerical defects in smart contracts, NumScout utilizes a symbolic execution framework to explore contract paths. A predefined semantic model assists in identifying execution states to capture key features and locate defects. In the following paragraphs, we provide a detailed explanation of how these defects are discovered in smart contracts.

(1) Div In Path: The tool first needs to locate the if comparison statements. Using the opcode sequence matching method mentioned above, it applies function cs to extract the comparison operator and two elements. The presence of this defect requires three conditions to be satisfied. First, the division operation must be on the left side of $>$ operator, and if subtraction is involved, the sides must be switched. Second, the division must be indivisible. Third, the two expressions being compared must contain user input values, meaning that the user input can influence the comparison result and thus affect the program's execution path. The reason why the first condition is necessary is that if the division operation appears on the left side of $<$, e.g., `if(a/100 < 3)`, then users may believe that the condition is satisfied when $a < 3 * 100$, which is indeed correct. The case where the division operation is on the right side of $>$ is discussed in Section III-C, where inexperienced users are more likely to be misled. The $\leq$ operator can be seen as an alternative branch of $>$, and both can be viewed as the same situation; similarly, $\geq$ and $<$ represent another equivalent situation.

Function E is used to extract the symbolic variables within the expressions. Function $i(a, b)$ checks whether a can divide b without leaving a remainder, which could lead to precision loss. The SMT solver adds this condition to the constraints and solves for the satisfiability. The detection rule for this defect is shown in Eq. 2.

(2) Operator Order Issue: The defect of computation order requires recovering the information of the expression $amount$ from $e(amount, v)$ to construct an operator order tree, which is accomplished by function bt. This function ensures that all operators are organized in a hierarchical structure, preserving their precedence as defined in the original expression. Then, a depth-first traversal of the tree is performed. This step is carried out by the dfs function, which takes the operator order tree returned by bt as input. It begins traversal from the root node and explores each node following a depth-first strategy,

$$amount = ea(S, M), \begin{cases} \dfrac{S :=< *, *, amount, *, *, *, * >}{ctx(transfer), CALL(f, *, *)} \\ \dfrac{amount := M[o + x], x > 4 \ \& \ x < l}{ctx(transfer), CALL(f, o, l), S :=< *, *, *, o, l, *, * >} \\ \dfrac{S :=< t, amount, arg2, ... >}{ctx(transfer), JUMP(t)} \end{cases} \quad (1)$$

Authorized licensed use limited to: Hangzhou Normal University. Downloaded on July 22,2025 at 02:27:38 UTC from IEEE Xplore. Restrictions apply.

inspecting the operator types along the way. If the pattern that division occurs before multiplication appears along any path, it indicates the presence of *Operator Order Issue*.

$$\frac{tree := bt(e(amount, v)), dfs(tree)}{ctx(transfer), amount = ea(S, M)}$$

(3) Minor Amount Retention: The defect of *Minor Amount Retention* requires not only that the transfer amount expression presents the possibility of being indivisible, but also that no other path exists to transfer the total ether or tokens held by the contract; otherwise, the retention defect does not exist. The expressions below illustrate our detection logic for *Minor Amount Retention*, where all possible paths for transferring ether and tokens are defined as $path$, and the path for transferring all ether or tokens is denoted as P.

$$\frac{(x, \div, y) := e(amount, v), i(x, y), \nexists P \in path}{ctx(transfer), amount = ea(S, M)}$$

(4) Exchange Problem: Our tool records the token flow in a structure t during the symbolic execution if it identifies a transfer operation. This is used to detect token exchange defects. Each flow consists of three elements: $from$, to, and $amount$. The function f is defined to find the two types of tokens involved in the exchange and their corresponding exchange amounts from structure t. From the contract's perspective, $in = 0 \wedge out \neq 0$ indicates that the user may gain profit for free. Conversely, the scenario $in \neq 0 \wedge out = 0$ indicates that rounding errors may occur during the exchange. Both conditions are passed to the SMT solver for checking. If the result is satisfiable, it can be confirmed that the *Exchange Problem* defect exists.

$$\frac{(in, out) := f(t), in = 0 \wedge out \neq 0 | in \neq 0 \wedge out = 0}{ctx(transfer)}$$

(5) Precision Loss Trend: To detect the defect of *Precision Loss Trend*, the function rt is first used to parse the expression and determine the rounding method. Specifically, it analyzes the operands and operators restored from the expression. If the numerator has been incremented by (denominator-1) before the division operation, the rounding type is identified as rounding up, i.e., $ceil(x)$. Then, the token flow's $from$, to, and rounding method are analyzed together using function ct. If the rounding method is $ceil(x)$ and the flow is outgoing from the contract, or if there are two or more flows with the same $from$ but different to and different rounding methods, the contract contains this defect. The former indicates that the direction of precision loss does not meet the requirements for maintaining the liquidity pool, while the latter suggests that there is unfair reward distribution.

$$\frac{r := rt(e(amount, v)), ct(from, to, r)}{ctx(transfer), amount = ea(S, M)}$$

TABLE II
FEATURES OF OUR DATASET VS. SMARTBUGS

Dataset \ Features	LOC	#of Instrs	#of Funs	#of State Vars
Ours	1155.9	8505.4	35.5	25.4
SmartBugs	99.9	1545.5	12.5	6.6

V. EXPERIMENT

In this section, based on an open-source dataset, we first conduct a small-scale experiment and evaluate the effectiveness of NumScout. We also perform detection on the large-scale dataset to confirm the situation of numerical defects in real-world contracts. Ablation experiments are conducted to demonstrate the effectiveness of the GPT-based pruning component.

A. Experimental Setup

The experiment is conducted on a server running Ubuntu 22.04.2 LTS, with a configuration of 20 Intel Xeon Platinum 8360H CPUs and 200GB of memory.

Dataset. To determine the prevalence of the defined defects in real-world Ethereum smart contracts, we utilize an open-source dataset from a GitHub repository [13], which stores the source code of all verified smart contracts on Etherscan up to July 13th, 2023. We downloaded this dataset on September 20th, 2024, and selected contract files deployed on the Ethereum mainnet, totaling 331,382 mainnet contracts. The dataset provides a summary file with basic information about each contract, e.g., contract's address, ether balance, compiler version, and total number of transactions. To filter for valuable contracts, we apply two criteria: total transactions > 100 and ether balance > 0. We further classify contracts by compiler version and remove those that cannot be compiled. These two filtering conditions ensure that the selected contracts are actively used by users in real-world scenarios, rather than toy contracts. This selection allows the experimental results to better reflect the tool's performance in detecting defects in widely used real-world contracts. Ultimately, we obtain 6,617 contracts.

We compare our dataset with SmartBugs [63], a widely used dataset. Table II highlights key characteristics of both datasets. It is evident that our dataset contains more complex smart contracts than those in SmartBugs. Specifically, the average lines of code (LOC) and the number of instructions of contracts in our dataset are **11.5X** and **5.5X** higher compared to SmartBugs, respectively. Additionally, the average number of public/external functions and state variables in our dataset are approximately **3X** and **4X** higher than in SmartBugs. 83.4% of the contracts in our dataset require a Solidity compiler version higher than v0.8.0, whereas 99.4% of the contracts in SmartBugs rely on versions below v0.5.0.

Evaluation Metrics. We outline the following research questions (RQs) to assess the effectiveness of NumScout.

$$\frac{T := E(x) \cup E(y), \exists s \in T, s \in Input, i(a, b)}{ctx(if), amount = ea(S, M), (x, > | \leq, y) := cs(S), (a, \div, b) = e(amount, v)} \quad (2)$$

Authorized licensed use limited to: Hangzhou Normal University. Downloaded on July 22,2025 at 02:27:38 UTC from IEEE Xplore. Restrictions apply.

TABLE III
DEFECTS IN SAMPLES AND EVALUATION OF NUMSCOUT

Defect	all	TP	FP	FN	Precision(%)	Recall(%)	F1-score(%)
Div In Path	7	7	0	1	100.0	87.5	93.3
Operator Order Issue	7	7	0	3	100.0	70.0	82.4
Minor Amount Retention	19	15	4	5	78.9	75.0	76.9
Exchange Problem	3	3	0	1	100.0	75.0	85.7
Precision Loss Trend	3	3	0	1	100.0	75.0	85.7

- **RQ1:** What is the efficacy of NumScout in detecting the five new types of defined numerical defects?
- **RQ2:** How effective is NumScout in detecting defects within our large-scale dataset?
- **RQ3:** How effective is the pruning component based on GPT?

```
function clearStuckBalance(uint256 amountPercentage)
    external onlyOwner {
    require(amountPercentage <= 100);
    uint256 amountBNB = address(this).balance;
    payable(_marketingWallet).transfer(amountBNB.mul
        (amountPercentage).div(100));}
```

Fig. 10. A FP case of minor amount retention defect.

B. Answer to RQ1: Evaluation of NumScout

To answer RQ1, we randomly sample a subset from the large-scale dataset for a small-scale experiment, where all samples are checked and labeled manually. Specifically, to determine the sample size, we follow a sampling method based on confidence intervals [64] to generalize the detection results from the sample to the overall dataset. We set a 10 confidence interval and 95% confidence level, calculating the required sample size to be 95. We randomly select a sample dataset and run NumScout on it. Two of the authors manually label the results of all samples carefully. We first collaboratively discuss and label 30% of the sample results to establish and confirm the labeling criteria. Then, we independently label the remaining 70% of the results, followed by a comparison and integration of the final results. We separate true positives (TP), false positives (FP), true negative (TN) and false negative (FN) during the labeling process to analyze the performance of NumScout. This method is also employed in other related works [65], [66], [67].

Table III displays the performance of NumScout on the labeled samples. The fifth to seventh columns show the number of TP, FP, FN in the samples, respectively. We use Precision $P = \frac{TP}{TP+FP}$, Recall $R = \frac{TP}{TP+FN}$, and F1-score $F1 = \frac{2*P*R}{P+R}$ to measure the detection performance for each type of defect. Additionally, we calculate the overall precision to demonstrate the effectiveness of NumScout. It can be calculated as $\frac{\sum_{i=1}^{n} p_{c_i} \times |c_i|}{\sum_{i=1}^{n} |c_i|}$, where p_{c_i} represents the precision of detecting defect i, and $|c_i|$ is the number of defect i in our dataset. NumScout achieves 100% precision in detecting the *Div In Path*, *Operator Order Issue*, *Exchange Problem* and *Precision Loss Trend*. For the *Minor Amount Retention*, it reports them at 78.9% precision. Overall, the comprehensive precision reaches 89.7%.

False Positives. Our experimental results contain some false positives in the detection of *Minor Amount Retention* due to an inability to recognize specific transfer paths. In certain contracts, users input a percentage number to withdraw funds from the contract as Fig. 10. The contract provides a $clearStuckBalance$ function, where the input parameter $amountPercentage$ represents the percentage of the total balance that the owner intends to withdraw. The function first ensures that the input $amountPercentage$ does not exceed 100%. It then calculates the withdrawal amount by multiplying the contract's total balance $amountBNB$ by the input percentage and dividing by 100. Finally, it transfers the computed amount to the designated wallet $_marketingWallet$. Users can withdraw funds at a 100% ratio, which creates a path that allows all funds to be transferred out. However, our tool fails to identify this special path for proportional fund withdrawal. Instead, due to its inability to track and analyze dynamic fund withdrawal conditions, it interprets the contract as potentially retaining a minor amount of funds. Consequently, this limitation leads to false positives when identifying the Minor Amount Retention defect.

False Negatives. We find that among the 95 samples, 11 are false negatives. All of the missed reports are caused by path explosion. Specifically, these contracts contain multiple branches in their CFG, leading to a huge search space. To avoid path explosion, we limit the tool's maximum loop iterations, the depth of path exploration, and the execution time. As a result, NumScout fails to detect the locations of these defects. It is worth noting that the main purpose of GPT-based pruning is to discard unrelated entry function paths, allowing the symbolic execution framework to reach the target function more quickly. However, it does not address the issue of overly deep search paths within the function.

For example, in one of the missed contracts[1], there are seven $require$ statements and five if conditional statements (lines 903-929) preceding the defective code, making the search paths extremely complex and causing NumScout to miss the defect as a consequence.

To mitigate false negatives caused by path explosion, the following optimization strategies can be considered. One approach is to introduce a heuristic search strategy. LLMs rank all functions based on their relevance to numerical operations and the

[1] aaf740FD71093520C457642eb9219A4F6dA22190

Authorized licensed use limited to: Hangzhou Normal University. Downloaded on July 22,2025 at 02:27:38 UTC from IEEE Xplore. Restrictions apply.

TABLE IV
DEFECTS IN LARGE-SCALE DATASET

Defect	# Defects	Percentage(%)
Div In Path	561	8.6
Operator Order Issue	306	4.7
Minor Amount Retention	983	15.1
Exchange Problem	39	0.60
Precision Loss Trend	114	1.7

```
1function exit() public {
2    if(_tokens > 0) sell(_tokens);
3    withdraw();}
4function withdraw() onlyStronghands() public {
5    uint256 _dividends = myDividends(false);
6    _customerAddress.transfer(_dividends);}
```

Fig. 11. A case of undetected operator order issue defect in the ablation experiment.

risk level of fund transfers, prioritizing the exploration of paths more likely to contain defects. LLMs can also integrate with dynamic symbolic execution to intelligently adjust subsequent search directions based on previously explored paths. Additionally, preprocessing complex control flow structures helps simplify the search space by flattening excessively nested loops and conditional branches where appropriate.

C. Answer to RQ2: Defects Detection in a Large-Scale Dataset

To address RQ2, we run NumScout on the source code of all the collected 6,617 verified smart contracts, which includes the 95 samples in RQ1. Table IV provides the numbers and frequency of each new type of numerical defect in contracts on Ethereum. NumScout only identifies whether a defect exists in the contract, so if the same type of defect appears multiple times, we count it only once.

Minor Amount Retention is the most common defect in our dataset, present in approximately 15.1% of the smart contracts. About 8.6%, 4.7%, and 1.7% of contracts contain the *Div In Path*, *Minor Amount Retention*, and *Precision Loss Trend* defects, respectively. Moreover, the proportions of *Exchange Problem* are all below 1%, with 39 (0.60%) smart contracts containing this defect.

Additionally, the experimental results indicate that 45 smart contracts contain 3 types of the 5 defined defects, while 194 smart contracts include 2 types of defects. Overall, as reported by NumScout, there are 1,774 smart contracts that have at least one type of defect in our dataset, which accounts for 26.8% of all contracts.

Contracts with *Minor Amount Retention* Defects. The large-scale experiment reveals that the number of contracts containing *Minor Amount Retention* defects is significantly higher than that of other defects. We find that many projects encounter cases where the profits cannot be divided evenly during distribution. Although, in the long term, the retained amount is only a small portion, represented as a random variable parameterized by the number of players, we consider that these small retained balances might be referenced by other contracts. Attackers may exploit this situation to inflict potentially substantial losses on other contracts.

D. Answer to RQ3: Ablation Experiment Results

In RQ3, we evaluate the effectiveness of the GPT-based pruning component. Specifically, we conduct an ablation experiment on the selected samples by removing the GPT-based pruning component. In this setup, the tool does not receive the list of functions unrelated to numerical operations or transfers, forcing it to explore all execution paths. The results show that the tool with pruning runs 28.4% faster than the version without pruning and identifies two additional *Operator Order Issue* defects. Specifically, to detect more defects, we set a time limit of 1,800 seconds and a search depth limit of 500 during the experiment. Additionally, we allow a longer SMT solver satisfiability checking time of 600 seconds at critical verification points. For each contract, the average runtime of the tool without the GPT-based pruning component reaches 1,518.56 seconds, while incorporating the GPT-based pruning component reduces the average runtime to 1,182.19 seconds. The average cost of the entire pruning process for a single contract on GPT-4o is only $0.008. Fig. 11 illustrates a defect detected in the RQ1 experiment but missed during the ablation experiment. The tool must execute the $sell$ function first, following specific paths to modify certain variables before the defect condition at line 6 is satisfied. The $sell$ function contains 9 function calls, 7 conditional statements, and about 40 numerical operations, resulting in a huge search space for symbolic execution.

Notably, our tool not only identifies the code location where the defect is triggered but also provides the entire call path, helping developers trace the defect's origin. From the results, we observe that the pruned version of the tool reaches the defect trigger point twice through different paths within the time limit. In contrast, without pruning, the tool wastes execution time in other unrelated functions, preventing symbolic execution from reaching the critical path within the time limit. The ablation experiment confirms that pruning enables the tool to enter target functions more quickly, improving detection speed and identifying more defects.

VI. DISCUSSION

A. Case Study

We present a real-world case[2] from the tool's report to illustrate how a user loses funds due to the new types of numerical defect and demonstrate the importance of detecting these defects reported by NumScout. Fig. 12 displays a simplified code snippet from the affected contract.

The user can purchase tokens by sending ether when calling the $sale$ function, but the token amount is calculated using a divide-then-multiply order. Given that the value of $cloudsPerEth$ on the current blockchain is 800,000, inexperienced users may assume that 1,000,000,000,000,000 wei

[2]0x3c07b3f4a6e253915d83c86707f0af07521d1cd8

Authorized licensed use limited to: Hangzhou Normal University. Downloaded on July 22,2025 at 02:27:38 UTC from IEEE Xplore. Restrictions apply.

```
1 function sale() payable {
2     uint256 amount = (msg.value / 1000000000000000)
          * cloudsPerEth;
3     balances[msg.sender] += amount;
4     balances[owner] -= amount;
5     Transfer(owner, msg.sender, amount);}
```

Fig. 12. Code snippet of the operator order issue and exchange problem case.

(i.e., 0.001 ether) is equivalent to 800,000 tokens, which means 1,250,000,000 wei is sufficient to buy 1 token. However, if the user sends less than 0.001 ether, the integer division results in $amount = 0$. Since the contract does not check the token amount exchanged, the transaction does not revert but continues to execute, leaving the users without any received tokens and causing them to lose the ether they sent. Our tool identifies this defect from two aspects: *Operator Order Issue* and *Exchange Problem*. The former is detected by analyzing the expression operator tree, while the latter is detected through token flow analysis.

We verify the funds loss process on the local test network with two test accounts. The first account deploys the contract, being the holder of the total token supply (i.e., $owner$), sets $cloudsPerEth$ to match its current value on the mainnet, and enables the trading switch. The second account (i.e., $msg.sender$) calls the $sale$ function, sending 0.0005 ether and expecting to receive 400,000 tokens. The result shows that while the ether balance of $msg.sender$ decreases, token balance remains 0. Meanwhile, the ether balance held by the contract increases by 0.0005 eth, and the $owner$'s token balance does not decrease. The verification script is available in our online repository.

B. Implications

For Researchers. Blockchains based on the EVM and supporting smart contract development in Solidity may exhibit similar numerical defects, though they may exhibit different patterns due to variations in blockchain characteristics. This possibility enables researchers to conduct further analysis and suggests new directions for future research.

For Practitioners. For developers, the defined defects aid in gaining a deeper understanding of numerical operations involved in smart contracts, particularly issues related to rounding and precision loss. It reminds developers to pay attention to minor precision losses and to improve testing efforts. These numerical defects can serve as coding guidance for developers during contract development to ensure robustness. For auditors, it enhances their awareness of the security about numerical operations, encouraging them to adopt more comprehensive auditing strategies.

For Investors and Users. It is important for investors and users to be cautious about potential numerical defects in contracts, which are often hidden within complex mathematical operations and can be difficult to detect. Additionally, our tool can help identify losses that may arise from numerical defects and in flagging contracts that might exploit these defects for fraudulent purposes.

For Educators. In smart contract development courses, educators should provide best practices and share known cases for avoiding numerical defects. This helps students recognize these defects and the serious consequences they may cause.

C. Threats to Validity

Internal Threats. One potential internal threat in our study is that we did not analyze all available audit reports, which may have led to the omission of some numerical defects. However, we mitigated this risk by utilizing an iterative information retrieval strategy to extract as many audit reports related to numerical defects as possible. The reports collected through this keyword-based approach help ensure comprehensive coverage and minimize the risk of missing relevant defects. Another internal threat arises from the high complexity of the smart contracts in our dataset, which makes the symbolic execution process highly time-consuming. Additionally, new types of numerical defects often involve division operations, which are computationally difficult for SMT solvers and require significant time to process. We address the execution time issue using GPT-based pruning, and ablation results confirm the effectiveness of this approach.

External Threats. Our dataset is filter based on specified criteria, which may have excluded numerical defects present in other contracts. However, by filtering contracts with more than 100 transactions and non-zero balances, our dataset reflects the numerical defects found in frequently used real-world contracts rather than those in test or toy contracts, providing a better evaluation of our tool's effectiveness. During the manual labeling process, there may be instances of incorrectly classifying false negatives and true negatives. To address this, we adopt a double-check mechanism and update the labeled dataset in a timely manner to ensure accuracy.

D. Possible Solutions for the Five Numerical Defects

In this subsection, we provide recommendations for developers to avoid introducing the defined five types of numerical defects in contracts. Section III presents defect code examples from audit reports, along with suggestions provided by security teams. We summarize the recommended fixes from the remaining audit reports, listing brief solutions for each type of defect in Table V. It is worth noting that in the *Operator Order Issue* defect, when changing the code from division before multiplication to multiplication before division, it requires careful consideration of overflow risks. For example, in the defective code shown in Fig. 12, if line 2 is modified to `uint256 amount = msg.value * cloudsPerEth / 1000000000000000`, it is important to note that $msg.value * cloudsPerEth$ might exceed the maximum value of $uint256$. As a result of the overflow, it will become a small number. This situation could occur when a user transfers a large amount of ETH to exchange for tokens, potentially causing losses of user funds. If the contract uses Solidity version v0.8.x, the compiler will automatically insert overflow checks into the bytecode, eliminating the need for developers to handle overflow risks. For versions lower than

Authorized licensed use limited to: Hangzhou Normal University. Downloaded on July 22,2025 at 02:27:38 UTC from IEEE Xplore. Restrictions apply.

TABLE V
POSSIBLE SOLUTIONS FOR THE FIVE DEFECTS

Defect	Possible Solution
Div In Path	Use multiplication instead of division in conditional statements.
Operator Order Issue	Multiply first and divide later, but be cautious of overflow.
Minor Amount Retention	Implement a function to withdraw all funds.
Exchange Problem	Check the calculation results before the transfer.
Precision Loss Trend	Consider who bears the loss of precision.
General Advice	Conduct thorough rounding tests. Avoid letting the liquidity pool bear precision loss. Ensure consistent precision between both tokens in the swap.

v0.8.0, developers should use the SafeMath library to prevent potential overflow.

Several additional suggestions for numerical operations are as follows: Developers need to thoroughly test rounding boundaries and rounding effects before deploying the contract. If there is a precision rounding issue in trading pool or lending pool, it is best not to let the liquidity pool bear the precision loss. Instead, calculations should favor the liquidity pool to ensure the pool remains balanced. Check the precision of the two assets being exchanged to prevent unexpected results due to differing precisions. For example, most ERC20 standard tokens have 18 decimals [49], while tokens like USDT have only 6 decimals. Some contracts do not handle these situation, leading to security problem. If calculations indeed involve assets with different precisions, prioritize using the asset with lower decimals for calculations. Then, derive the amount of the asset with higher decimals through multiplication. This way, all mathematical operations are based on multiplication, avoiding the creation of decimal units. To validate the effectiveness of these solutions, we randomly select 10 contracts for each type of defect and apply the recommended fixes. We then analyze these revised contracts using NumScout, and the results show that NumScout reports no defects.

VII. RELATED WORK

A. Smart Contract Defects

Chen et al. propose the first research that defines smart contract defects from the developers' perspective [43]. They collect posts from StackExchange and use an open card sorting method to discover and categorize 20 types of contract defects. Additionally, they design a survey to gather developers' feedback and concerns regarding these defects. In another work, they introduce a tool named DefectChecker [68], which detects these defined defects by analyzing the bytecode of contracts. However, their research do not cover the new types of numerical defect that arise in smart contracts. Specifically, they defines $UnmatchedTypeAssignment$, which focuses on mismatches between assignments and types, potentially leading to integer overflow. This differs from the new numerical defects we focus on, which can result in transaction execution errors.

B. Tools for Smart Contract Defects Detection

Many program analysis tools focus on detecting traditional numerical defects. Luu et al. proposed the first symbolic execution-based tool, Oyente [65], which simulates EVM instruction execution and explores different execution paths to construct CFG. It uses the Z3 SMT solver to determine whether vulnerability conditions are satisfied, enabling the detection of overflow vulnerabilities. Torres et al. introduced a framework called Osiris [69] that identifies three types of integer-related defects in Ethereum smart contracts through taint analysis: Arithmetic Bugs, Truncation Bugs, and Signedness Bugs. Additionally, other static analysis tools such as MAIAN [70], Securify [71], Ethainter [72], Sailfish [73], Mythril [74], and Slither [75] have also been developed to detect defects in Solidity smart contracts. Meanwhile, tools like ContractFuzzer [66], sFuzz [76], Smartian [77], and Echidna [78] are based on dynamic testing and analysis.

C. Accounting Errors

Another type of smart contract defect involving numerical operations is the Accounting Errors. They are specific to the financial logic of the contract, focusing on incorrect financial logic operations, such as adding fees to a user's balance instead of deducting them or directly summing tokens of different units. The tool ScType [79] models financial operations and high-level information in DeFi, e.g., token units, scaling factors, and financial types, and leverages type propagation and checking to detect Accounting Errors. ScType relies on specific business contexts and requires manual completion of initial type annotations. In contrast, our work complements this work by focusing on issues arising from the nature of numerical calculations themselves, such as precision loss and improper operator order, which may lead to unexpected behaviors during smart contract execution.

D. LLMs in Smart Contract Defect Detection

Currently, LLMs are widely used in smart contract defect detection. Sun et al. propose GPTScan [35], the first tool that integrates GPT with static analysis for detecting logical vulnerabilities in smart contracts. This tool uses GPT to identify key variables and statements, followed by static analysis to verify potential vulnerabilities. Ma et al. introduce the iAudit framework [36], which combines LLM fine-tuning with a multi-role strategy to audit contracts through iterative debates. Ding et al. propose SmartGuard [37], a framework that retrieves semantically similar code, generates CoT, and then utilizes LLMs for vulnerability identification. Wang et al. present ContractTinker [38], which also employs CoT and program static analysis to guide LLMs in repairing real-world smart contract vulnerabilities. Wu et al. develop AdvSCanner method [39], which uses static analysis to extract attack flows related to reentrancy vulnerabilities and utilizes them to guide LLMs in generating attack contracts that can exploit reentrancy vulnerabilities in victim contracts. Our work uses LLMs for pruning and combines symbolic execution tools to confirm new types of numerical defects, which expands and complements these existing works.

Authorized licensed use limited to: Hangzhou Normal University. Downloaded on July 22,2025 at 02:27:38 UTC from IEEE Xplore. Restrictions apply.

VIII. Conclusion

There are two main parts in this paper: the definition of defects and their detection. We summarize five new types of numerical defect patterns from the audit reports provided by the DAppScan dataset, which are collected from multiple renowned blockchain security teams. These issues are considered high-risk and impact the execution results of programs. For each defect, we provide code examples and possible solutions. To identify defects in real-world smart contracts, we develop a tool called NumScout, which utilizes GPT-based pruning and symbolic execution to detect the aforementioned five defined defects.

NumScout uses GPT-4o for pruning, removing functions unrelated to numerical operations and transfers, thus enhancing the efficiency of subsequent symbolic execution. The tool performs symbolic execution at the bytecode level, combined with source code features for analysis. Specifically, the tool constructs and analyzes expression operator order tree, extract the conditional statements of comparison from the bytecode, analyzes token flows, and other methods to extract key features. It reports defects based on predefined defect patterns combined with source code mapping. Moreover, NumScout supports all compiler versions and is extensible, allowing developers to write additional detection patterns to identify more defects. Experimental results show that NumScout identifies 1,774 smart contracts containing at least one defined defect in the dataset. Furthermore, NumScout achieves an overall detection precision of 89.7%.

References

[1] V. Buterin et al., "A next-generation smart contract and decentralized application platform," *White Paper*, vol. 3, no. 37, pp. 2–1, 2014.

[2] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: A survey," *Int. J. web Grid Services*, vol. 14, no. 4, pp. 352–375, 2018.

[3] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "An overview of blockchain technology: Architecture, consensus, and future trends," in *Proc. IEEE Int. Congr. Big Data (BigData Congr.),* Piscataway, NJ, USA: IEEE Press, 2017, pp. 557–564.

[4] "Solidity programming language," 2024. Accessed: Oct. 17, 2024. [Online]. Available: https://soliditylang.org/

[5] M. R. Lyu et al., *Handbook of Software Reliability Engineering*. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1996, vol. 222.

[6] XBlock, "Smart contract defects-arithmetic issue," 2024. Accessed: Oct. 17, 2024. [Online]. Available: https://xblock.pro/#/article/55

[7] OpenZeppelin, "The safemath library for solidity smart contracts," 2020. Accessed: Oct. 17, 2024. [Online]. Available: https://docs.openzeppelin.com/contracts/3.x/api/math

[8] Ethereum Foundation, "Solidity v0.8.0 breaking changes, solidity 0.8.0 documentation," 2020. Accessed: Oct. 17, 2024. [Online]. Available: https://docs.soliditylang.org/en/v0.8.0/080-breaking-changes.html

[9] "Balancer," 2024. Accessed: Oct. 17, 2024. [Online]. Available: https://docs.balancer.fi/

[10] BlockSec, "Tiny rounding down, big fund losses: An in-depth analysis of the recent balancer incident," 2023. Accessed: Oct. 17, 2024. [Online]. Available: https://blocksec.com/blog/tiny-rounding-down-big-fund-losses-an-in-depth-analysis-of-the-recent-balancer-incident

[11] D. Spencer, *Card sorting: Designing Usable Categories*. Rosenfeld Media, 2009.

[12] J. Wei et al., "Emergent abilities of large language models," 2022, *arXiv:2206.07682*.

[13] M. Ortner and S. Eskandari, "Smart contract sanctuary." Accessed: Oct. 17, 2024. [Online]. Available: https://github.com/tintinweb/smart-contract-sanctuary

[14] smlXL, "Evm codes - an ethereum virtual machine opcodes interactive reference," 2024. Accessed: Oct. 17, 2024. [Online]. Available: https://www.evm.codes/

[15] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (SOK)," in *Princ. Secur. Trust: 6th Int. Conf., POST, Held as Part Eur. Joint Conferences Theory Pract. Softw., ETAPS*, Uppsala, Sweden. Berlin, Heidelberg: Springer, 2017, pp. 164–186.

[16] Z. Wang, H. Jin, W. Dai, K.-K. R. Choo, and D. Zou, "Ethereum smart contract security research: Survey and future research opportunities," *Frontiers Comput. Sci.*, vol. 15, no. 2, pp. 1–18, 2021.

[17] Z. A. Khan and A. S. Namin, "A survey on vulnerabilities of ethereum smart contracts," 2020, *arXiv:2012.14481*.

[18] N. F. Samreen and M. H. Alalfi, "A survey of security vulnerabilities in ethereum smart contracts," 2021, *arXiv:2105.06974*.

[19] "Bec token," 2018. Accessed: Oct. 17, 2024. [Online]. Available: https://etherscan.io/address/0xc5d105e63711398af9bbff092d4b6769c82f793d#code

[20] "Smartmesh," 2018. Accessed: Oct. 17, 2024. [Online]. Available: https://etherscan.io/address/0x55f93985431fc9304077687a35a1ba103dc1e081#code

[21] "Uselessethereumtoken," 2018. Accessed: Oct. 17, 2024. [Online]. Available: https://etherscan.io/address/0x27f706edde3ad952ef647dd67e24e38cd0803dd6#code

[22] "Openzeppelin," 2024. Accessed: Oct. 17, 2024. [Online]. Available: https://www.openzeppelin.com/

[23] W. X. Zhao et al., "A survey of large language models," 2023, *arXiv:2303.18223*.

[24] Y. Chang et al., "A survey on evaluation of large language models," *ACM Trans. Intell. Syst. Technol.*, vol. 15, no. 3, pp. 1–45, 2024.

[25] "Openai," 2024. Accessed: Oct. 17, 2024. [Online]. Available: https://openai.com/

[26] A. Vaswani, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017.

[27] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 35, 2022, pp. 22199–22213.

[28] Openai, "gpt-4o," 2024. Accessed: Oct. 17, 2024. [Online]. Available: https://platform.openai.com/docs/models

[29] C. Chen et al., "When chatGPT meets smart contract vulnerability detection: How far are we?" 2023, *arXiv:2309.05520*.

[30] J. Chen, C. Chen, J. Hu, J. Grundy, Y. Wang, T. Chen, and Z. Zheng, "Identifying smart contract security issues in code snippets from stack overflow," in *Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2024, pp. 1198–1210.

[31] W. Ma et al., "Unveiling code pre-trained models: Investigating syntax and semantics capacities," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 7, pp. 1–29, 2024.

[32] Z. Zheng et al., "Towards an understanding of large language models in software engineering tasks," *Empirical Softw. Eng.*, vol. 30, no. 2, p. 50, 2025.

[33] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an LLM to help with code understanding," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, 2024, pp. 1–13.

[34] Y. Zhang, "Detecting code comment inconsistencies using llm and program analysis," in *Companion Proc. 32nd ACM Int. Conf. Found. Softw. Eng.*, 2024, pp. 683–685.

[35] Y. Sun et al., "GPTSCAN: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, 2024, pp. 1–13.

[36] W. Ma et al., "Combining fine-tuning and LLM-based agents for intuitive smart contract auditing with justifications," 2024, *arXiv:2403.16073*.

[37] H. Ding, Y. Liu, X. Piao, H. Song, and Z. Ji, "SMARTGUARD: An LLM-enhanced framework for smart contract vulnerability detection," *Expert Syst. with Appl.*, vol. 269, 2025, Art. no. 126479.

[38] C. Wang, J. Zhang, J. Gao, L. Xia, Z. Guan, and Z. Chen, "Contracttinker: Llm-empowered vulnerability repair for real-world smart contracts," in *Proc. 39th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2024, pp. 2350–2353.

[39] Y. Wu et al., "Advscanner: Generating adversarial smart contracts to exploit reentrancy vulnerabilities using LLM and static analysis," in *Proc. 39th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2024, pp. 1019–1031.

[40] Z. Zheng, J. Su, J. Chen, D. Lo, Z. Zhong, and M. Ye, "DAppSCAN: Building large-scale datasets for smart contract weaknesses in DApp

Authorized licensed use limited to: Hangzhou Normal University. Downloaded on July 22,2025 at 02:27:38 UTC from IEEE Xplore. Restrictions apply.

projects," *IEEE Trans. Softw. Eng.*, vol. 50, no. 6, pp. 1360–1373, Jun. 2024.
[41] "Consensys," 2024. Accessed: Oct. 17, 2024. [Online]. Available: https://consensys.io/
[42] L. A. Goodman, "Snowball sampling," *Anna. Mathematical statistics*, pp. 148–170, 1961.
[43] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining smart contract defects on ethereum," *IEEE Trans. Softw. Eng.*, vol. 48, no. 1, pp. 327–345, Jan. 2020.
[44] Solidity, "Division — solidity 0.8.29 documentation," 2024. Accessed: Oct. 17, 2024. [Online]. Available: https://docs.soliditylang.org/en/latest/types.html#division
[45] "Chainsecurity," 2024. Accessed: Oct. 17, 2024. [Online]. Available: https://www.chainsecurity.com/
[46] "Quillaudits," 2024. Accessed: Oct. 17, 2024. [Online]. Available: https://www.quillaudits.com/
[47] "Dedaub," 2024. Accessed: Oct. 17, 2024. [Online]. Available: https://dedaub.com/
[48] "Trail of bits," 2024. Accessed: Oct. 17, 2024. [Online]. Available: https://www.trailofbits.com/
[49] V. B. Fabian Vogelsteller, "Erc-20: Token standard," 2015. Accessed: Oct. 17, 2024. [Online]. Available: https://eips.ethereum.org/EIPS/eip-20
[50] "Usdt," 2024. Accessed: Oct. 17, 2024. [Online]. Available: https://tether.to/
[51] "Usdc," 2024. Accessed: Oct. 17, 2024. [Online]. Available: https://www.circle.com/en/usdc
[52] "Xrp," 2024. Accessed: Oct. 17, 2024. [Online]. Available: https://www.ibcprotocol.dev/
[53] Etherscan.io, "Token tracker," 2024. Accessed: Oct. 17, 2024. [Online]. Available: https://etherscan.io/tokens
[54] "Peckshield," 2024. Accessed: Oct. 17, 2024. [Online]. Available: https://peckshield.com/
[55] "Uniswap," 2024. Accessed: Oct. 17, 2024. [Online]. Available: https://uniswap.org
[56] S. Yang, J. Chen, and Z. Zheng, "Definition and detection of defects in nft smart contracts," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2023, pp. 373–384.
[57] "ethereum/go-ethereum," 2024. Accessed: Oct. 17, 2024. [Online]. Available: https://github.com/ethereum/go-ethereum
[58] Solidity, "Source mappings — solidity 0.8.29 documentation," 2024. Accessed: Oct. 17, 2024. [Online]. Available: https://docs.soliditylang.org/en/latest/internals/source_mappings.html
[59] L. De Moura and N. Bjórner, "Z3: An efficient smt solver," in *Int. Conf. Tools Algorithms Construction Anal. Syst.*, Springer, 2008, pp. 337–340.
[60] J. He, G. Sivanrupan, P. Tsankov, and M. Vechev, "Learning to explore paths for symbolic execution," in *Proc. 2021 ACM SIGSAC Conf. Comput. Commun. Secur.*, 2021, pp. 2526–2540.
[61] L. Huang et al., "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *ACM Trans. Inf. Syst.*, vol. 43, no. 2, pp. 1–55, 2025.
[62] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "LLM is like a box of chocolates: The non-determinism of chatGPT in code generation," 2023, *arXiv:2308.02828*.
[63] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "SmartBugs: A framework to analyze solidity smart contracts," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2020, pp. 1349–1352.
[64] Wikipedia, "Confidence interval," 2024. Accessed: Oct. 17, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Confidence_interval
[65] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 254–269.
[66] B. Jiang, Y. Liu, and W. K. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," in *Proc. 33rd ACM/IEEE Int. Conf. automated Softw. Eng.*, 2018, pp. 259–269.
[67] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *Proc. NDSS*, 2018, pp. 1–12.
[68] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "DefectChecker: Automated smart contract defect detection by analyzing EVM bytecode," *IEEE Trans. Softw. Eng.*, vol. 48, no. 7, pp. 2189–2207, Jul. 2022.
[69] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 664–676.
[70] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 653–663.
[71] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 67–82.
[72] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: A smart contract security analyzer for composite vulnerabilities," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2020, pp. 454–469.
[73] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves, "Sailfish: A framework for large scale data processing," in *Proc. 3rd ACM Symp. Cloud Comput.*, 2012, pp. 1–14.
[74] Mythril, "Mythril," 2023. Accessed: Oct. 17, 2024. [Online]. Available: https://mythril-classic.readthedocs.io/en/master/module-list.html
[75] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proc. IEEE/ACM 2nd Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB),* Piscataway, NJ, USA: IEEE Press, 2019, pp. 8–15.
[76] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, 2020, pp. 778–788.
[77] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, "SMARTIAN: Enhancing smart contract fuzzing with static and dynamic data-flow analyses," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE),* Piscataway, NJ, USA: IEEE Press, 2021, pp. 227–239.
[78] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: effective, usable, and fast fuzzing for smart contracts," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2020, pp. 557–560.
[79] B. Zhang, "Towards finding accounting errors in smart contracts," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, 2024, pp. 1–13.

Authorized licensed use limited to: Hangzhou Normal University. Downloaded on July 22,2025 at 02:27:38 UTC from IEEE Xplore. Restrictions apply.